Algorithm Analysis Part 2

Complexity Analysis
Introduction

Algorithm Analysis measures the efficiency of an algorithm, or its implementation as a program, as the input size becomes large. Actually, an estimation technique and does not tell anything about the relative merits of two programs. However, it does serve as a tool for us to determine whether an algorithm is worth considering.
Introduction

Typically, we will analyze the time required for an algorithm and the space required for a data structure. Primary concern, the number of basic operations to process an input of a certain size. A basic operation has the property that its time to complete does not depend on the values of its operands.
Growth Rate

The **growth rate** for an algorithm is the rate at which the cost of the algorithm grows as the size of its input grows.

- **Linear** – grows as a straight line
- **Quadratic** – grows as a quadratic
- **Exponential** – grows exponentially
Best, Worst and Average Cases

Best Case – If the data is arranged in such a way that when the algorithm runs it looks at the fewest data items.

Worst Case – If the data is arranged in such a way that when the algorithm runs it looks at the most data items.

Average Case – If the algorithm is run many times, with random ordering of the data, the average number of data items the algorithm examines.
Asymptotic Analysis

Asymptotic Analysis refers to the study of an algorithm as the input size reaches a limit.

We use some simplifying notions:
- Big O
- Big Omega
- \( \Theta \) Notation
The upper bound for an algorithm is used to indicate the upper or highest growth rate.

We will measure this upper bound with respect to the best, worst or average case.

We say “this algorithm has an upper bound to its growth rate of $f(n)$ in the average case”.

Or we say the algorithm is “in $f(n)$”.
Precise Definition

$T(n)$ represents the running time of the algorithm

$f(n)$ is some expression for the upper bound

For $T(n)$ a non-negatively valued function, $T(n)$ is in set $O(f(n))$ if there exists two positive constants $c$ and $n_0$ such that $T(n) \leq cf(n)$ for all $n > n_0$. 
The lower bound for an algorithm is used to indicate the lowest growth rate. We will measure this lower bound with respect to the best, worst or average case. This is known as big Omega or just Omega.
Precise Definition

$T(n)$ represents the running time of the algorithm

$f(n)$ is some expression for the lower bound

For $T(n)$ a non-negatively valued function, $T(n)$ is in set $\Omega(f(n))$ if there exists two positive constants $c$ and $n_0$ such that $T(n) \geq cf(n)$ for all $n > n_0$
An algorithm is said to be $\Theta(h(n))$ if it is in $O(h(n))$ and if it is $\Omega(h(n))$.

$\Theta$ notation is a stronger statement to say because it requires information about both the upper and lower bounds.

It should be used when information about both the upper and lower bounds is available.

Sometimes it is easier to express information about a particular instantiation of an algorithm than the algorithm itself.
Simplifying Rules

1. If some function is an upper bound for your cost function, then its upper bound are also upper bounds for your function.
2. You can safely ignore any multiplicative constants.
3. When given two parts of a program run in sequence, you need consider only the more expensive part.
4. If an action is in a loop, and each repetition has the same cost, then the total cost is the cost of the action multiplied by the number of times the action takes place.
Calculating the Running Time of a Program

\[ a = b; \]
  \[- \Theta(1)\]
\[ \text{sum} = 0; \]
\[ \text{for (i=1; i<=n; i++)} \]
  \[ \text{sum} += n; \]
  \[- \Theta(n)\]
\[ \text{sum} = 0; \]
\[ \text{for (j=1; j<=n; j++)} \]
  \[ \text{for (i=1; i<=j; i++)} \]
    \[ \text{sum}++; \]
\[ \text{for (k=0; k<=n; k++)} \]
  \[ A[k] = k; \]
  \[- \Theta(n^2)\]
Common Misunderstandings

- Upper and lower bounds are only interesting when you have incomplete knowledge about the thing being measured.
- The upper bound is not the same as the worst case.
- We don’t want to tie best and worst cases to input size.
Space Bounds

Primary purpose of a data structure is...
- To store data!!!

Often you must store additional information in the data structure to allow access to the data

This additional information is called overhead
The Space/Time Tradeoff

The space/time tradeoff principle says that one can often achieve a reduction in time if one is willing to sacrifice space or vice versa.

Lookup tables allow the program to pre-store information that it will need often for easy lookup at the expense of needing to store the table.

Let’s not even consider if this information is stored on disk, talk about a tradeoff in space and time to access a disk.